Data Structures

Dr Ahmed Rafat Abas
Computer Science Dept, Faculty of Computer and Information, Zagazig University
arabas@zu.edu.eg
http://www.arsaliem.faculty.zu.edu.eg/

Searching Algorithms

Chapter 11

11.1 The Study Of Algorithms

- An algorithm is a set of instructions for achieving a particular goal.
- There are several different algorithms for the same operation that is performed on data structures.
- To choose the best algorithm for a certain problem, you need to know the running time and memory requirements.
- The theoretical study of algorithms is known as **complexity theory** which involves analysis of the number of steps required by an algorithm for varying amount of data.

11.2 Sequential Search

- A searching algorithm requires a target for which to search.
- The list of data is searched until either
  - the target is located or
  - the algorithm determines that the target is not in the list.
- For primitive data types as int and float, a simple comparison is performed between the target and the list of data.

- For user defined data types, one of the fields is chosen as the key field.
- The target is then compared only to the key field.
- In sequential search, the algorithm starts with the first element of the list and compare each element with the target.
  - The index of the corresponding element is returned if the search is successful.
  - Flag value as -1 is returned if the search fails.

The sequential search algorithm

1. Read in Target

2. Initialize Marker to first list index(0)

3. While Target != Key at location Marker and not at the end of the list :

   Marker++.

4. If Target == Key at location Marker,

   then return Marker.

   else, return -1.

11.3 The C++ code for Sequential Search

- The base class called datalist is defined which is a general purpose array-based class containing :

  - Dynamic array Element,

  - a constructor,

  - a destructor,

  - and input/output functions.

- The class datalist is a template class, so any type of data can be searched.

- In the datalist class the operators << and >> are overloaded for input and output , so the user-defined objects can be read from ifstream objects (such as cin) and written to ofstream objects (such as cout).

datalist.h

The definition of the class is in the file datalist.h

```
#ifndef DATALIST_H
#define DATALIST_H
#include<iostream.h>

template<class Type>
class datalist  {
protected:
      Type *Element;
      int ArraySize;
public:
      datalist(int arraysize=10):ArraySize(arraysize),
                  Element(new Type[arraysize]){}
      virtual ~datalist();
      friend ostream& operator<<(ostream& OutStream,const
         datalist<Type>& Outlist);
      friend istream& operator>>(istream& InStream,
        datalist<Type>& Intlist);
};
#endif
```

search.h

The class searchlist inherits datalist and adds a Search() function which implements the sequential search algorithm. This code in the file search.h

```
#ifndef SEARCH_H
#define SEARCH_H
#include "datalist.h"

template <class Type>
class searchlist : public datalist<Type> {
public:
   searchlist (int arraysize = 10) :
        datalist<Type>(arraysize) {}
   virtual ~searchlist() {}
   virtual int search(const Type& Target) const;
 };
#endif
```

The definitions of the functions in the header file datalist.h can be written in the file datatemp.h as follows :

```
#ifndef DATATEMP_H
#define DATATEMP_H
#include “datalist.h”

template <class Type>
datalist<Type>::~datalist(){Delete [] Element;}

template <class Type>
ostream& operator<<(ostream& OutStream,const
datalist<Type>& OutList)
{
OutStream << “Array contents:\n”;
for(int element=0;element<OutList.ArraySize;element++)
   OutStream << OutList.Element[element]<<‘ ‘ ;
OutStream << endl;
OutStream << “ArraySize: “ << OutList.ArraySize<<endl;
return OutStream;
}
```

```
template <class Type>
istream& operator>>(istream & InStream,datalist<Type>
& InList)
{
cout << “Array contents:\n”;
for(int element=0;element<InList.ArraySize;element++)
{
   cout<< “Element “ << element << “:” ;
   InStream >> InList.Element[element];
}
return InStream;
}
#endif
```

The definition of the **Search()** function which is in the header file search.h can be written in the file searchtm.h as follows :

```
#ifndef SEARCHTM_H
#define SEARCHTM_H
#include “search.h”
#include “datatemp.h”

template <class Type>
int searchlist<Type>::search(const Type& Target) const
{
for(int element=0;element<ArraySize;element++)
{
   if(Element[element]==Target)
      return element;
}
return -1;
}
#endif
```

A **main()** function that tests these routines is in the file search.cpp

```
#include "searchtm.h"
const int SIZE = 5 ;
main() {
searchlist<float> List1(SIZE);
float Target;
int Location;
cin>> List1;
cout<<List1;
for(int i= 0; i < 5 ; i++){
   cout<< " search for a float:  ";
   cin>>Target;
   if((Location = List1.search(Target)) != -1 )
      cout<<" Found at index "<<Location<<endl;
   else
      cout<< "Not Found.\n";
}
return 0;
}
```

11.4 Binary Search

- Binary search can be:
  - Forgetful search which doesn't check if the target is found until the end of the algorithm.
  - Target search which checks if the target is found at each step.
- In binary search, the list is chopped to half at each step.
- Each step we have half keys to be searched.
- In binary search, three markers are used, (top , bottom and middle)

- **The binary search algorithm (forgetful search) can be performed on a list of numdata elements as follows:**

1. Set `top=numdata-1` ; set `bottom = 0`.
2. While `top>bottom` do:
   3. set `middle=(top+bottom)/2` [using integer division]
   4. if key at index `middle` is less than target key,
      then set `bottom = middle +1`
      else, set `top = middle`.
5. If `top == -1`,
   return -1 indicating that the list is empty.
6. If key at location `top` = the target key,
   then, return location `top` indicating that the target has been found.
   else, return -1 indicating target has not been found.

Forgetful Binary Search

1. Set `top=numdata-1` ; set `bottom = 0`.
2. While `top>bottom` do:
   3. set `middle=(top+bottom)/2` [using integer division]
   4. if key at index `middle` is less than target key,

      then set `bottom = middle +1`

      else, set `top = middle`.
5. If `top == -1`,

   return -1 indicating that the list is empty.
6. If key at location `top` = the target key,

   then, return location `top` indicating that the target has been found.

   else, return -1 indicating target has not been found.

0
1
2
3
4
5

Step	1	2	3	4
Bottom	0	3	3	3
Middle	2	4	3	3
Top	5	5	4	3

11.5 The C++ code for Binary search

The class forgetsearch inherits the base class datalist and adds a Search() function which implements the binary search algorithm. This code is in the file binary.h

```
#ifndef BINARYH
#define BINARYH
#include "datalist.h"

template <class Type>
class forgetsearch : public datalist<Type> {
public:
   forgetsearch(int arraysize=10):
      datalist<Type>(arraysize){}
   virtual ~forgetsearch() {}
   virtual int Search(const Type& Target) const;
};
#endif
```

The definition of the Search() function that implements the forget binary search is defined in the file bintemp.h as follows:

```
#ifndef BINTEMP
#define BINTEMP
#include “binary.h”
#include “datatemp.h”
Template <class Type>
int forgetsearch<Type>::Search(const Type& Target) const {
int top=ArraySize-1, bottom=0, middle;
while (top>bottom){
   middle=(top+bottom)/2;
   if (Element[middle] < Target)
      Bottom=middle+1;
   else
      Top=middle;
}
if (top==-1)
   return -1;
if(Element[top] == Target)
   return top;
else
  return -1;
}
```

```
// this function implements the Target binary search
Template <class Type>
int targetsearch<Type>::Search(const Type& Target) const
{
int top=ArraySize-1, bottom=0, middle;
while(top>=bottom){
   middle=(top+bottom)/2;
   if (Element[middle] == Target)
      return middle;
   else if(Element[middle] < Target)
      Bottom=middle+1;
   else
      top=middle-1;
}
return -1;
}
#endif
```

11.6 Searching Algorithms Efficiency

- There are three cases for every algorithm in which its efficiency is measured:
  - the **best case,**
  - **the average** case
  - **the worst** case.
- In sequential search,
  - the best case occurs when the target is the first item in the list, so only one comparison is made.
  - the worst case is either the target is the last item or it is not in the list.
  - the average case is found by considering all possible outcomes of a search and averaging the number of all comparisons over all these cases.

11.7 Sequential Search Efficiency

- The best case of sequential search is one comparison and it is independent of the list length n
- The worst case requires n comparison where the target is the last item or not exist.
- The worst case is proportional to the list length n.
- The average case can be obtained by adding up the number of comparisons for all possible outcomes and then dividing by the number of possible outcomes.

- The search outcomes can be separated to ***successful*** and ***unsuccessful*** outcomes of the search.
- For an unsuccessful search, it requires n comparisons, so the best, average and worst efficiencies are all the same.
- For a successful search, the best case requires one comparison and the worst case require n comparison. Average case of a successful search require n/2 comparisons where we expect that the half of the list is searched before the target is found.

11.8 Binary Search Efficiency

- Binary search looks first at the middle.
- If the target lies in the upper half of the list, the lower half is ignored.
- If the target lies in the lower half of the list, the upper half is ignored.
- For a list whose length is (2^n), there will be $n+1$ comparisons.
- The binary search efficiency is proportional to n for a list whose length=2^n.